CORE JAVA MODULE 4 NOTES

01 July 2025 11:52

➢ Module 4: Arrays and Strings

No.	Topic
4.1	1D Arrays – Declaration, Initialization, Traversal
4.2	2D Arrays – Matrix-style operations , Jagged Array and Anonymous Array
4.3	Array Operations – Sorting, Searching, Sum, Max
4.4	Strings – String Class, StringBuilder, StringBuffer
4.5	Important String Methods – length, substring, indexOf, split, etc.
4.6	Mutable vs Immutable Strings
4.7	Interview Questions and Assignment

⇒ Module-4.1: 1D Arrays – Declaration, Initialization, Traversal

✧ Real-Life Analogy:

🧦 **Imagine a drawer with 5 boxes, each holding one pair of socks.**

- Each box can only hold **one type** (e.g., socks, not socks + shoes).
- You can access any box using a **number** — box 0, box 1, ..., box 4.
- You know upfront how many boxes (size is fixed).

◆ This is how a **1D Array** works in Java — a fixed collection of same-type elements, stored in a row, accessible by index.

Concept:

- Arrays store **multiple values of the same data type** in a **single variable**.
- Useful when you want to store a list of items like marks, names, prices, etc.
- Indexing starts from **0** and goes to **length - 1**.

Syntax + Explanation:

1 Declaration Only:

```
int[] marks;    // OR  int marks[];
```

✧ This just creates a reference; no memory is allocated yet.
✧ A **reference variable marks** is created in the **stack memory**.

2 Declaration + Memory Allocation

```
marks = new int[5];  // Allocates space for 5 integers
```

✧ Java allocates memory for **5 integers** in the **heap memory** (each int = 4 bytes × 5 = 20 bytes).
✧ All elements are initialized to **0** by default.
✧ The address (reference) of this array in heap is stored in marks.

✧ Now you can access marks[0] to marks[4]

3 Declaration + Allocation + Initialization

```
int[] marks = {85, 90, 78, 88, 95};
```

✧ A short and quick way to assign values.

Traversal of Array
Accessing elements one by one is called **traversal**.

Using **For Loop**

```
for (int i = 0; i < marks.length; i++) {
    System.out.println("marks[" + i + "] = " + marks[i]);
}
```

Using **Enhanced For Loop (For-each)**

✧ **What is Enhanced For Loop (For-each)?**

Real-Life Analogy:

Imagine a teacher passing **exam papers** to each student in the class **one by one**, without worrying about their roll numbers or positions.

You don't care **which number student** you're on — just go through each paper directly.
That's exactly what **Enhanced For Loop** does — it visits every element in the array **one by one**, without using an index.

Definition:
The **enhanced for loop** (also called **for-each loop**) is used to **traverse through arrays or collections** in a clean and simple way — without needing to manage the index manually.

Syntax:

```
for (dataType variable : arrayName) {
    // use variable
}
```

```
for (int m : marks) {
    System.out.println(m);
}
```


Use Cases:

- When you only want to **access values**, not modify them.
- Perfect for **read-only traversals**.

Limitations:

- Cannot **skip** or **go backward**.
- Cannot **access index** like arr[i].
- Cannot **change values** inside the array directly.

Example Demo:

```
public class ArrayDemo {
    public static void main(String[] args) {
        int[] marks = {85, 90, 78, 88, 95};
```

```
        System.out.println("All Marks:");
        for (int i = 0; i < marks.length; i++) {
            System.out.println("Subject " + (i + 1) + ": " + marks[i]);
        }
    }
}
```

Output:

```
All Marks:
Subject 1: 85
Subject 2: 90
Subject 3: 78
Subject 4: 88
Subject 5: 95
```

Common Mistakes:

1. **Accessing out of bounds** index
   marks[5] → ArrayIndexOutOfBoundsException

2. **Using different data types** in one array
   int[] arr = {10, "hello", 3.14}; → Not allowed!

3. **Modifying array size later**
   marks.length = 10; → Not possible. Arrays are fixed-size.

✧ Advantages and Disadvantages of Arrays

Advantage	Explanation
1. Easy Access	Elements can be accessed instantly using index (arr[i]) – O(1) time.
2. Fixed Size	You know exactly how much memory to allocate.
3. Performance	Very fast for searching, iterating, summing when size is fixed.
4. Clean Code	Instead of 10 separate variables, just use one array variable.
5. Index-Based	Easy to loop, sort, or search using simple logic.

Disadvantage	Explanation
1. Fixed Size	Once declared, size can't be changed.
2. Wasted Memory	If array is bigger than needed, unused memory is wasted.
3. Manual Work	You need to manually sort, search, insert, delete — no built-in methods like in ArrayList.
4. Same Type Only	All elements must be of the same type (int, String, etc).
5. Risk of Error	ArrayIndexOutOfBoundsException is a common bug for beginners.

Real-Life Use Cases of Arrays

Real-Life Scenario	How Array is Used
Student Marks	Store marks of students in an exam: int[] marks = new int[50];
Sensors	Store last 100 temperature readings in a smart device
Stock Market	Store stock prices for the past 7 days
Quiz App	Store correct answers: String[] answers = {...}

String Characters	char[] ch = word.toCharArray(); to manipulate individual characters

➢ **Arrays are best when:**

- The number of items is **known in advance**
- Data is **of same type**
- You want **fast access** via index

When Not to Use Array?

If your size is **not fixed** or you want dynamic resizing, **use**:

- ArrayList
- LinkedList
- HashMap (for key-value pairs)

⇒ Module 4.2 – 2D Arrays: Matrix-style Operations

Real-Life Analogy

Think of your **college seating arrangement**.

- 4 rows and 3 columns.
- Each seat has a student.
- To locate a student → go to **row i and column j**.

Just like that, **2D Arrays** store data in a grid format (rows × columns).

2D array = Array of Arrays

Concept

A **2D Array** is like a **table or matrix**:

- Rows: Horizontal line
- Columns: Vertical line

Syntax + Declaration

1 Declaration Only

```
int[][] matrix;
```

2 Declaration + Memory Allocation

```
matrix = new int[3][4]; // 3 rows, 4 columns
```

3 Declaration + Allocation + Initialization

```
int[][] matrix = {
        {10, 20, 30},
        {40, 50, 60}
};
```

Memory Explanation:

```
int[][] matrix = new int[3][4];
```

- matrix is a reference in **stack**

- Actual array rows live in **heap**
- Each row is itself a **1D array**

matrix[0] is an array → [0, 0, 0, 0]

✧ Traversal (Nested Loop)

```
for (int i = 0; i < matrix.length; i++) {          // Rows
    for (int j = 0; j < matrix[i].length; j++) {   // Columns
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

✧ Example Program

```
public class TwoDArrayDemo {
    public static void main(String[] args) {
        int[][] matrix = {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}
        };

        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
1  2  3
4  5  6
7  8  9
```

✧ Jagged Array (Irregular Rows)

A **Jagged Array** is a 2D array with rows of **different lengths**.

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];  // 2 elements
jagged[1] = new int[4];  // 4 elements
jagged[2] = new int[3];  // 3 elements
```

You can even do:

```
int[][] jagged = {
        {1, 2},
        {3, 4, 5, 6},
        {7, 8, 9}
};
```

- Each row is still a separate array — hence they can be different lengths.

✧ Anonymous Arrays

An **anonymous array** is created without a reference — useful when passing directly to a method:

```
new int[] {10, 20, 30}
```

Example:

```
public static void printSum(int[] arr) {
    int sum = 0;
    for (int n : arr)
        sum += n;
    System.out.println("Sum = " + sum);
}
```

Advantages of 2D Arrays

- Clean way to store tabular data
- Easy access via indexes
- Memory efficient for fixed-size grids

Disadvantages

- Fixed size → can't dynamically grow
- Tedious to sort/search (unless nested logic is written)
- Hard to insert/delete rows/columns

✧ Real-Life Use Cases

Scenario	How 2D Array is Used
◆ Student Marks	◆ Store marks of 4 students in 5 subjects: int[4][5]
◆ Matrix Math	◆ Matrix multiplication, transpose, determinant
◆ Excel Sheets	◆ Store row × column data

⇒ Module 4.3 – Array Operations: Sorting, Searching, Sum, Max:

Real-Life Analogy

- **Sum**: Adding up your grocery bill.
- **Searching**: Finding your name in an attendance list.
- **Max**: Looking for the highest score in a class.
- **Sorting**: Arranging students in order of marks.

These are **array operations** — done on lists of values to extract or organize information.

1. Sum of Elements

```
int[] arr = {10, 20, 30, 40};
int sum = 0;

for (int num : arr) {
    sum += num;
}

System.out.println("Sum = " + sum);
```

2. Find Maximum (or Minimum)

```
int[] arr = {10, 90, 30, 80};
int max = arr[0];

for (int i = 1; i < arr.length; i++) {
    if (arr[i] > max) {
        max = arr[i];
    }
}

System.out.println("Max = " + max);
```

Similarly, use < to find minimum.

3. Searching in Array (Linear Search)

```
int[] arr = {10, 25, 50, 60};
int key = 25;
boolean found = false;

for (int i = 0; i < arr.length; i++) {
    if (arr[i] == key) {
        found = true;
        System.out.println("Found at index " + i);
        break;
    }
}

if (!found)
    System.out.println("Not Found");
```

This is called **Linear Search** — goes element by element.

4. Sorting the Array (Ascending):

Using Arrays.sort() (Inbuilt)

```
import java.util.Arrays;

int[] arr = {30, 10, 20, 50, 40};
Arrays.sort(arr);

System.out.println(Arrays.toString(arr));
```

5.Manual Sorting (Bubble Sort : Friendly)

```
int[] arr = {5, 3, 4, 1, 2};

for (int i = 0; i < arr.length - 1; i++) {
    for (int j = 0; j < arr.length - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
            // Swap
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
```

```
System.out.println(Arrays.toString(arr));
```

✧ Anonymous Array

Imagine you want to give a parcel to someone but don't want to keep it for later — just hand it over directly.
new int[]{1, 2, 3} is like handing over the parcel directly, without storing it in a cupboard (no variable).

Syntax : new int[] {1 ,2};

```
public class Demo {

   // Method that takes array input and prints sum
   static void printSum(int[] arr) {
      int sum = 0;
      for (int i : arr) {
         sum += i;
      }
      System.out.println("Sum is: " + sum);
   }

   public static void main(String[] args) {
      // Anonymous array passed directly
      printSum(new int[]{10, 20, 30, 40});
   }
}
```

1. Array without name .
2. Created and Initialized in a single line.
3. It can be single or multi-dimensional.
4. It can be used only once.
5. It can be used an argument in method.

⇒ Module 4.4 – Strings – String Class, StringBuilder, StringBuffer

Real-Life Analogy

	Java Type	Why?
◆ **Sealed envelope** – once you glue it, contents can't change. If you want new text you must prepare *another* envelope.	◆ **String**	◆ Immutable – every change creates a brand-new object.
◆ **Whiteboard** – you can erase and rewrite freely.	◆ **StringBuilder**	◆ Mutable, fast, single-thread use.
◆ **Whiteboard with a safety lock** – only one person can write at a time.	◆ **StringBuffer**	◆ Mutable **and** thread-safe via synchronization.

✧ What *is* a String?

- String is not primitive datatype, because its size can't fix.
- String is the sequence of characters.
- String is the array of characters.
- String is a class
- Proper Syntax of String is -> public final class String
- String class extends Object class and implements CharSequence , Serializable , Comparable
- A **sequence of characters** stored as an **object** of class java.lang.String.
- **Immutable**: any change (concatenation, replace, substring…) returns a **new** object.

Why immutability?

String Pool – JVM can safely share identical literals to save memory.

✸ String Literals vs String Objects (new keyword)

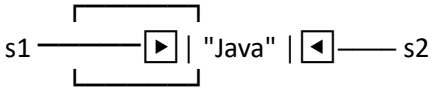
Real-Life Analogy

Type	Real-Life Analogy	What It Means
◆ **Literal**	◆ Like a shared WhatsApp group message — if the same message already exists, Java says: *"Don't create a new one, just use the existing one."*	◆ Reuses existing memory from the **String Pool**
◆ **Object (new)**	◆ Like printing a fresh newspaper each time, even if the news is same — wastes resources	◆ Always creates **new memory** in Heap, even if same text

A. String Literal (Stored in String Constant Pool)

```
String s1 = "Java";
String s2 = "Java";
```

What Happens in Memory?



(Stored in String Pool)

- **Only one object** is created in **String Constant Pool**
- Both s1 and s2 point to the **same memory location**
- JVM checks **if already exists**, reuses it
- Memory-efficient, fast access

B. String Object using new keyword (Stored in Heap)

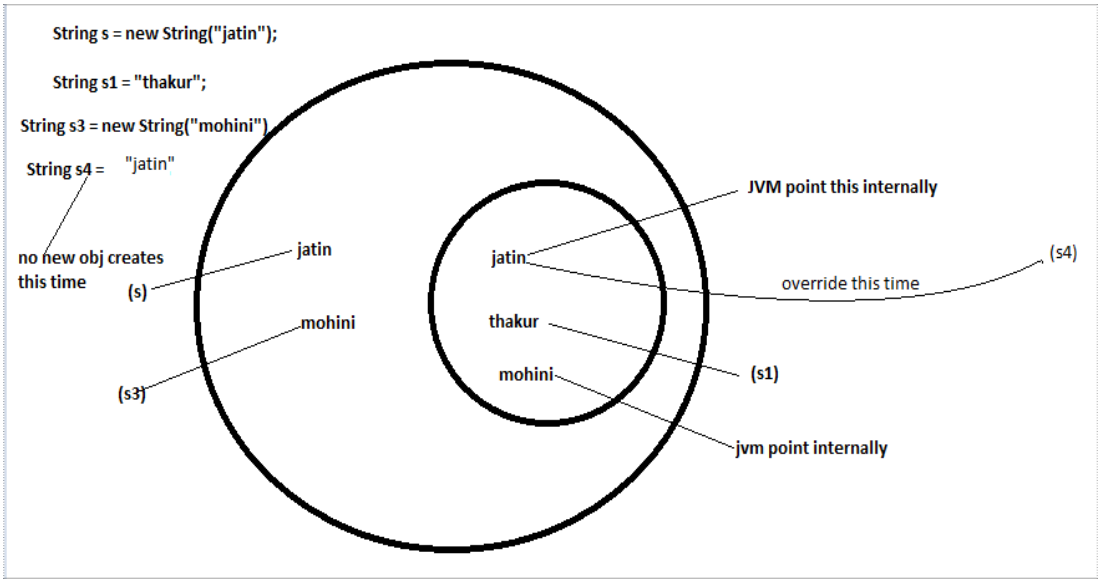
```
String s3 = new String("Java");
String s4 = new String("Java");
```

What Happens in Memory?

- JVM **does not reuse** even if content is same
- **New object** is created in Heap every time
- Involves **more memory**, **slower comparisons**
- Needed only when **you intentionally want a new instance**

Disadvantages of new String("...")

Issue	Why it matters
Extra memory	Creates new object even if same value exists in pool
Slower	Equals/hashCode must be recalculated
More garbage	Multiple unused "Java" objects floating in heap



- **String Constant Pool :(inside heap area):**

Q1. String Immutability ?
Q2. Why String objects are immutability?

A1. Immutability means unchangeable .

- Immutability concept is use for String objects , String objects are immutable .it means once String object is created : its data and state can't be changed but a new string object is created.

A2. Understand through example :

- String city = "mohali" -> for jatin
- String city2 = "mohali" -> for rashid
- String city 3 = "mohali" -> for neeraj

In this only one object is created in the Sting constant pool and all the city will refer to this **mohali**

Advantage : only one object create this time and our project execution should be faster .

- Now Assume that neeraj shifted towards Chandigarh then it will affect to jatin and rashid, and it is a big issue . So that's why String objects are immutable.

⇒ Module 4.5 – Important String Methods in Java

1. length()

Definition:
Returns the number of characters in the string.

What it does:
Counts spaces, letters, symbols – everything.

```
String s = "Java World";
System.out.println(s.length());  // 10
```

2. charAt(index)

Definition:
Returns the character at the specified position (0-based index).

```
String s = "Code";
System.out.println(s.charAt(1));  // o
```

3. substring(startIndex) and substring(startIndex, endIndex)

Definition:
Returns part of the string.

What it does:
Cuts a portion starting from startIndex to end (or to endIndex - 1).

```
String s = "Programming";
System.out.println(s.substring(3));     // gramming
System.out.println(s.substring(3, 7));  // gram
```

4. indexOf(char or String)

Definition:
Returns the index of first occurrence of a character or substring.

Returns:

- Index if found
- -1 if not found

```
String s = "Hello Java";
System.out.println(s.indexOf('J'));      // 6
System.out.println(s.indexOf("Java"));   // 6
System.out.println(s.indexOf("Z"));      // -1
```

5. lastIndexOf(char or String)

Definition:
Finds the last position of the given character/string.

```
String s = "banana";
System.out.println(s.lastIndexOf("a"));  // 5
```

6. replace(old, new)

Definition:
Replaces every occurrence of the old character or string with new one.

```
String s = "cool boot";
System.out.println(s.replace("o", "x"));  // cxxl bxxt
```

7. equals(str) and equalsIgnoreCase(str)

Definition:

- equals() → case-sensitive comparison

- equalsIgnoreCase() → ignores case while comparing

```
String a = "Hello";
String b = "hello";

System.out.println(a.equals(b));           // false
System.out.println(a.equalsIgnoreCase(b)); // true
```

8. contains(substring)

Definition:
Returns true if the string contains the specified substring.

```
String s = "JavaScript";
System.out.println(s.contains("Script"));  // true
```

9. startsWith() and endsWith()

Definition:
Checks if string starts or ends with a given prefix/suffix.

```
String s = "DataStructure";
System.out.println(s.startsWith("Data"));  // true
System.out.println(s.endsWith("ure"));     // true
```

10. trim()

Definition:
Removes leading and trailing spaces (not inside words).

```
String s = "   Java   ";
System.out.println(s.trim());  // "Java"
```

11. toUpperCase() and toLowerCase()

Definition:
Converts the string to all uppercase or lowercase letters.

```
String s = "Java";
System.out.println(s.toUpperCase());  // JAVA
System.out.println(s.toLowerCase());  // java
```

12. concat(str)

Definition:
Joins two strings (same as + operator).

```
String a = "Hello";
String b = "World";
System.out.println(a.concat(" " + b));  // Hello World
```

13. split(delimiter)

Definition:
Splits string into array of substrings using a delimiter.

```
String s = "apple,banana,grapes";
        String[] fruits = s.split(",");

        for (String f : fruits) {
            System.out.println(f);
        }
// apple
// banana
// grapes
```

14. isEmpty() and isBlank() *(Java 11+)*

Method	Returns true if…
isEmpty()	length is 0
isBlank()	contains only spaces or is empty

```
String a = "";
String b = "   ";

System.out.println(a.isEmpty());  // true
System.out.println(b.isEmpty());  // false
System.out.println(b.isBlank());  // true
```

15. valueOf(data)

Definition:
Converts any data type (int, float, char, etc.) into a String.

```
int age = 25;
String s = String.valueOf(age);  // "25"
```

16. compareTo() and compareToIgnoreCase()

Definition:
Compares strings lexicographically.

Return	Meaning
0	Both strings are equal
< 0	First string comes before
> 0	First string comes after

```
System.out.println("Apple".compareTo("Banana"));  // -1 or negative
System.out.println("abc".compareToIgnoreCase("ABC"));  // 0
```

Core Difference

Feature	Comparable	Comparator

Package	java.lang	java.util
Method	compareTo()	compare()
Changes in	Class being compared	Separate class (or lambda)
Used for	Natural/default ordering	Custom/specific ordering

Real-Life Analogy:

Think of Comparable like a **student who always compares based on roll number** (his natural way of comparison).
Think of Comparator like an **external examiner who compares students by marks or names or height** (custom logic).

Student Class with Comparable (Default: Compare by rollNo)

```
import java.util.*;
class Student implements Comparable<Student> {
    int rollNo;
    String name;
    int marks;
    Student(int rollNo, String name, int marks) {
        this.rollNo = rollNo;
        this.name = name;
        this.marks = marks;
    }
    // Natural ordering: by roll number
    public int compareTo(Student s) {
        return this.rollNo - s.rollNo;
    }
    public String toString() {
        return rollNo + " " + name + " " + marks;
    }
}
Usage:

public class TestComparable {
    public static void main(String[] args) {

        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Jatin", 90));
        list.add(new Student(1, "Amit", 85));
        list.add(new Student(2, "Neha", 95));

        Collections.sort(list); // Uses compareTo
        System.out.println("Sorted by rollNo (Comparable):");
        for (Student s : list) {
            System.out.println(s);
        }
    }
}
```

Now Comparator: Sorting by Marks, then Name

```
 // Comparator to sort by marks (descending)
        class SortByMarks implements Comparator<Student> {
            public int compare(Student s1, Student s2) {
                return s2.marks - s1.marks;
            }
        }
// Comparator to sort by name
        class SortByName implements Comparator<Student> {
            public int compare(Student s1, Student s2) {
                return s1.name.compareTo(s2.name);
            }
        }
```

Usage:

```
public class TestComparator {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Jatin", 90));
        list.add(new Student(1, "Amit", 85));
        list.add(new Student(2, "Neha", 95));
        System.out.println("\nSorted by Marks (Comparator):");
        Collections.sort(list, new SortByMarks());
        for (Student s : list) System.out.println(s);
        System.out.println("\nSorted by Name (Comparator):");
        Collections.sort(list, new SortByName());
        for (Student s : list) System.out.println(s);
    }
}
```

Shortcut Summary:

- Use Comparable when default sorting logic should be inside the class (like sort by rollNo).
- Use Comparator when:
  - You want **multiple sorting options** (marks, name, etc).
  - You can't or don't want to change the original class.

✧ Why StringBuilder and StringBuffer?

Because **String is immutable**, every change creates a **new object**, which is memory-wasting in loops or dynamic changes.
So Java gave us **mutable** versions:

- StringBuilder (faster, for single-threaded apps)
- StringBuffer (slower, but thread-safe)

Definitions

Type	Definition
◆ StringBuilder	◆ A class to **create mutable strings** — best when you need to modify strings repeatedly in single-threaded apps.
◆ StringBuffer	◆ Same as StringBuilder, but **thread-safe** (synchronized) — used when multiple threads access same string.

Syntax & Examples

🔷 1. StringBuilder Example

```
public class Demo {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
        sb.append(" Java");
        sb.insert(0, "Hey! ");
        sb.replace(0, 4, "Hi");
        sb.delete(3, 5);
        System.out.println(sb);  // Output: Hi Java
    }
}
```

🔷 2. StringBuffer Example

```java
public class Demo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Code");
        sb.append(" with Fun");
        System.out.println(sb);  // Output: Code with Fun
    }
}
```

✧ Common Methods

Method	What It Does
.append("text")	Adds text to end
.insert(pos, "txt")	Inserts at given index
.replace(start, end, "txt")	Replaces between start-end
.delete(start, end)	Deletes part
.reverse()	Reverses string
.capacity()	Shows current buffer capacity
.toString()	Converts to normal String

✧ Difference: StringBuilder vs StringBuffer

Feature	StringBuilder	StringBuffer
Mutability	Mutable	Mutable
Thread Safety	Not thread-safe	Thread-safe
Performance	Faster	Slower (synchronized)
Use-case	Single-threaded apps	Multi-threaded apps
Introduced	Java 1.5	Java 1.0

```java
public class Compare {
    public static void main(String[] args) {
        long start, end;

        // Using String
        start = System.currentTimeMillis();
        String s = "";
        for (int i = 0; i < 10000; i++) {
            s += i;
        }
        end = System.currentTimeMillis();
        System.out.println("String: " + (end - start) + "ms");

        // Using StringBuilder
        start = System.currentTimeMillis();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 10000; i++) {
            sb.append(i);
        }
        end = System.currentTimeMillis();
        System.out.println("StringBuilder: " + (end - start) + "ms");
    }
}
```

✧ Recap

Feature	String	StringBuilder	StringBuffer
◆ Mutable?	◆ No	◆ Yes	◆ Yes
◆ Thread-safe?	◆ No	◆ No	◆ Yes
◆ Fast?	◆ Slow (in loops)	◆ Fastest	◆ Medium
◆ Use in loops?	◆ Avoid	◆ Yes	◆ Yes (if multi-threaded)

Module 4.7 – Interview Questions and Assignments

✧ Beginner-Level Interview Questions

No.	Question	Answer
1	◆ What is an array?	◆ A collection of elements of the same type stored in a contiguous memory block.
2	◆ Can we change the size of an array once declared?	◆ No, arrays in Java are fixed-size.
3	◆ What is the default value of an int array element?	◆ 0 for int, false for boolean, null for objects.
4	◆ What is the index of the first element in an array?	◆ Index starts from 0.
5	◆ What is an enhanced for loop?	◆ A simplified loop to iterate over arrays or collections without using index.
6	◆ What is the difference between length and length()?	◆ length is a property of arrays; length() is a method for strings.
7	◆ Can arrays hold different data types?	◆ No, arrays are type-specific (e.g., int[], String[]).
8	◆ What is a 2D array?	◆ An array of arrays (matrix-style), like int[][] matrix.
9	◆ What is a String literal?	◆ A string value directly written in code (e.g., "Hello").
10	◆ Are strings mutable in Java?	◆ No, String is immutable – cannot be changed once created.

✧ Intermediate-Level Interview Questions

No.	Question	Answer
1	◆ How is memory allocated for arrays?	◆ Reference stored in Stack, actual elements in Heap.
2	◆ Difference between StringBuilder and StringBuffer?	◆ Both mutable, but StringBuffer is thread-safe.
3	◆ What is ArrayIndexOutOfBoundsException?	◆ Exception thrown when accessing invalid index in array.
4	◆ What is a jagged array?	◆ A 2D array where inner arrays have different lengths.
5	◆ How to find the max element in an array?	◆ Loop through array and compare elements using if.
6	◆ Can you reverse a string in Java?	◆ Yes, using StringBuilder and its .reverse() method.
7	◆ Difference between == and .equals() for strings?	◆ == compares memory reference, .equals() compares content.
8	◆ What does split(",") do in a string?	◆ Breaks string into array using , as separator.
9	◆ Can we pass an array to a method?	◆ Yes, by passing array reference (int[] arr).
10	◆ What is the use of .charAt(index)?	◆ Returns the character at a specific position in a string.

✧ Expert-Level Interview Questions

No.	Question	Answer
1	◆ How does Java's String Pool work?	◆ JVM stores string literals in a common pool to save memory.
2	◆ Why is String immutable in Java?	◆ For security, thread-safety, caching (hashCode), and performance.

3	◆ What is the internal structure of a 2D array in Java?	◆ It's an array of references to 1D arrays (not real matrix).
4	◆ How is capacity managed in StringBuilder?	◆ Starts with 16; expands as needed: (oldCapacity * 2) + 2.
5	◆ Can you explain substring memory leak (pre-Java 7)?	◆ Substrings shared char[] with original → caused memory leaks.
6	◆ Difference between .equals() and .compareTo()?	◆ .equals() checks equality; .compareTo() gives sorting order.
7	◆ Is StringBuilder faster than + in loops?	◆ Yes, avoids creating multiple objects, more efficient.
8	◆ What happens if null is passed to .equals()?	◆ It throws NullPointerException if used like null.equals(...).
9	◆ How is an anonymous array declared?	◆ new int[]{1, 2, 3} – no name, used directly.
10	◆ Can arrays be resized at runtime?	◆ No, you must create a new array and copy elements.

Assignments & Coding Practice

No.	Task
1	Write a program to count vowels in a string using charAt() and length()
2	Find the max and min in an integer array
3	Sort an array without using built-in sort (bubble sort or selection sort)
4	Reverse a string using StringBuilder.reverse()
5	Check if a string is a palindrome
6	Create a 2D array for a 3x3 matrix and print its transpose
7	Find the frequency of each character in a string
8	Remove duplicates from a string using logic (no Set)
9	Merge two arrays into one sorted array
10	Split a string by space and print each word on a new line